EFFICIENT DETECTION OF HANG BUGS IN MOBILE APPLICATIONS

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By
Deepa Thiagarajan
Graduate Program in Electrical and Computer Engineering

The Ohio State University
2016

Master's Examination Committee:
Dr. Xiaorui Wang, Advisor
Dr. Joanne E. DeGroat
Abstract

Mobile applications (apps) are growing tremendously in the market. Many apps are useful for our day-to-day activities. In order to survive in the market, these apps must be highly responsive, and free of bugs. For an app to be popular among users, developers must focus on its responsiveness. If the response time of an app is more than the human perceivable delay (100 ms), then the user will feel the app to be slow or unresponsive. This is called the ‘hang bug’ problem. Developers must ensure that apps are programmed well so as to detect and correct hang bugs.

Most of the existing work on hang bugs in mobile apps uses the offline detection technique i.e., detecting bugs before an app is released in the market. Hence, some bugs might exist at runtime when the user interacts with the app. If the offline technique is used, the hang bug problem will remain unknown to the developers until after release, when it will be too late to change. In this thesis, an efficient algorithm has been proposed to detect hang bugs in mobile apps during runtime, as opposed to the offline technique. The algorithm efficiently detects the API/method that is the root cause of the hang bug. There are six algorithms proposed to efficiently detect hang bugs with least delay, memory and CPU utilization. The various cases were tested on real world apps, and results were compared to find the most efficient of the cases for implementation.

Key Words: Android Applications, Hang Bug, Algorithm, Delay, CPU, Memory Utilization
This document is dedicated to my husband and son
Acknowledgments

I would like to thank my advisor, Dr. Xiaorui Wang, for giving me a lot of freedom in completing my thesis. I have learned from him professionally and also personally, in becoming a dedicated researcher. I have not only developed my research skills from Dr. Wang but I have also learned how to stay calm and focused during my thesis journey. I feel honored to have him as my research advisor.

To Dr. Joanne DeGroat, I express my heartfelt gratitude in agreeing to serve on my Masters’ thesis committee, and going through the drafts to offer helpful suggestions for improvement.

I am indebted to Mr. Marco Brocanelli for his constant support and help throughout my Master’s research. He gave me lots of valuable suggestions on the algorithm design and implementation.

I am grateful to my husband and son for supporting me throughout this journey. Without my husband’s help and my son’s encouragement, I wouldn’t have been able to get my Master’s degree.

I am forever in gratitude to my parents who have always been wishing for my success and taught me the necessary values in life to succeed in my endeavors.
Vita

2007…………………B.S., Electronics and Communication Engineering

Pondicherry University,
Pondicherry, India

2009…………………MSc., Wireless Sensors and Embedded System

University of Bradford
Bradford, United Kingdom

2014-Present………M.S., Electrical and Computer Engineering

The Ohio State University
Columbus, Ohio, U.S.A

Field of Study

Major Field: Electrical and Computer Engineering
# Table of Contents

Abstract ........................................................................................................................................... ii

Dedication ........................................................................................................................................ iii

Acknowledgments ............................................................................................................................ iv

Vita .................................................................................................................................................... v

List of Tables ..................................................................................................................................... ix

List of Figures .................................................................................................................................... x

Chapter 1: Introduction .................................................................................................................... 1

Chapter 2: Literature Survey and Related Work ........................................................................... 3

Chapter 3: Background .................................................................................................................... 5

  3.1: Android .................................................................................................................................... 5

  3.2: Android Applications .............................................................................................................. 5

  3.3: Hang Problem ......................................................................................................................... 6

  3.3.1: Reasons for Hang ................................................................................................................ 6

  3.4: Soft Hang ................................................................................................................................ 7

Chapter 4: Algorithm Design ........................................................................................................... 9
6.4: Whole App Overhead ................................................................. 36

6.5: Conclusions ........................................................................ 38

References .................................................................................... 39
List of Tables

Table 1. Efficiency Comparison Chart ................................................................. 38
List of Figures

Figure 1. Event Detector in Main Thread (Synchronized Thread Sleep Method)........... 14
Figure 2. Event Detector in Background Thread (Synchronized Thread SleepMethod).................................................................................................................... 16
Figure 3. Event Detector in Main Thread (Wait and Notify Method and Timer Task in the Background Thread)............................................................................ 18
Figure 4. Event Detector in the Background Thread (Wait and Notify Method and Timer Task) .............................................................................................................................................. 20
Figure 5. Timer Task in Event Detector and Both in the Main Thread....................... 22
Figure 6. Timer Task in Event Detector and Both in the Background Thread............... 24
Figure 7. Timer Task in Event Detector and Both in the Background Thread.............. 31
Figure 8. Delay Plot ......................................................................................................... 33
Figure 9. Main Thread Overhead Plot ........................................................................... 35
Figure 10. WholeApp Overhead Plot............................................................................... 37
Chapter 1: Introduction

In recent years, smartphone applications (shortened as ‘apps’ throughout this document) have been growly tremendously. In order for the apps to stay in the market, smartphone application developers must mainly focus on the quality of the application. If the application is being coded in an inefficient way then it freezes, hangs, stops or becomes unresponsive. The application is considered to hang or act unresponsive if the reaction time of the app is longer than the human perceivable delay (100ms) [1]. For example, whenever an app displays a delay of more than 100ms for user inputs such as scroll, push buttons etc. [1], the application is considered to hang. The mechanism for the hang bug problem can be described by understanding the roles played by the main thread and background thread, in android apps. The main thread in the android apps is responsible for handling and processing user inputs. Whenever the main thread receives a user input, it executes the code for a certain amount of time before it handles the next user input. The main thread must always complete the user input within 100ms (human perceivable delay) [1]. If the main thread takes more than 100ms to complete the task, then the user might perceive the app to be unresponsive or slow. Therefore, heavy operations must always be performed in a different thread called the background thread instead of the main thread, in order to avoid the hang bug problem.
A large body of research exists in order to find different hang bug problems in android applications. But, most of the conducted research utilizes an offline detection technique, which means trying to find hang bug problems before the app is released in the market. As a result, there are a lot of hangs bugs unknown to the developer at runtime, but which become evident when the user interacts with the app after the app has been released in the market (Google Play store). Also, most of the research work on detecting hang bugs at runtime is focused on servers and desktops as opposed to the more frequently used mobile phones, which are also severely affected by hang bug problems.

In this thesis, an efficient algorithm has been proposed to detect hang bugs caused during run time whenever the user interacts with the app. This algorithm detects the heavy operations that were running on the main thread and those that were missed by the offline detection technique. It also monitors the response time of each long-running events, and the resource usage of the main thread. Hence, these heavy operations that were running on the main thread are detected efficiently at run time when the user interacts with the app, and information is relayed to the developer for rectifying such errors during the future development of app.

The thesis is organized as follows. Chapter 2 focuses on existing literature and related work in detecting hang bugs in apps. Chapter 3 discusses the background of android and the different causes of soft hang. Chapter 4 concentrates on the design of the algorithm. In Chapter 5, the algorithm is executed and evaluated for several test apps, and its efficiency is highlighted. Chapter 6 describes the conclusions drawn from this thesis work. The bibliography section at the end provides a list of references, which have influenced the current thesis and the algorithm implementation.
The most important factor to be considered when developing an app is the responsiveness of the app. The user might eventually be forced to uninstall the app if the app becomes slow and unresponsive. As a result, this might affect the sales and ratings of the app, and stall the future development of that app. To address this concern, a good amount of research has been ongoing in finding the root causes for hang bugs. Most of the existing research has utilized the offline hang bug detection technique i.e., detecting hang bugs before the app is released in the market. Yu Lin et al [2]. designed an asynchronizer, which is an automatic refactoring tool, which will enable the developer to extract the long running operations into AsynTask [2]. Since the heavy operations might block the main thread, the entire heavy job is being done in the AsynTask or background thread. Y. Liu et al. [3] conducted a study on real world performance bugs, and they proposed a technique in order to detect these bugs statically. The authors in [3] concluded the fact that these long running operations, which are performed on the main thread, can make the app unresponsive. X.Wang et al. [4] tried to find common reasons that can cause hangs in desktop and server applications. Yang et al. [5] tried to find the reasons for the unresponsiveness of different Android apps by adding a very long delay after each heavy API (Application Program Interface). X. Song et al. [6] conducted a study on the characteristics of the hang bugs. He conducted it from the four common open source
applications and tried to provide guidelines in order to fix them. S. Yang et al. [7] developed a tool, which will automatically detects all the long running or heavy operations on the main thread in the applications in order to avoid the hang problem.

Most of the existing researches works have utilized the offline detection of hang bugs, i.e., detecting hang bugs before the app is released in the market. Although these techniques can be useful for the developer to detect hang bugs before releasing the app in the market, these algorithms fail to take into consideration two major problems, namely false positive and false negative. False positive means the algorithm warns the developer regarding a hang bug but this bug does not cause any long delay and also the user will not perceive any delay during operation of the app. False negative implies that the delay is not detected by offline detection algorithms since there are large number of different new APIs, and it is difficult to have a complete knowledge of these APIs. The behavior of these different and new APIs also changes, depending on the device type, environment and OS version.

Therefore the proposed algorithm in this thesis helps to detect the hang bugs at runtime when the user interacts with the app. The algorithm developed and implemented in this thesis mainly helps detect the long running APIs that were missed by the offline detection tools.
Chapter 3: Background

3.1 Android

Android is an operating system which is being developed by Google for different smartphones and tablets. The operating system which is developed by Google is based on Linux kernel and it provides a set of APIs (Application Program Interface) in order to access different underlying features and services. Google always maintains an open source operating system so that everyone can build and install these apps and can also make changes to it. [8]

3.2 Android Applications

The android applications are also known in their abbreviated form as apps. Users can get these applications from Google Play or the Amazon app store for day today activities. The android device always has the Google Play store app pre-installed on it in order to access Google Play store and to download these apps. There are more than billions of apps present in the Google Play store app. [9].

Often times, when a user installs an app from the Google Play store, and when the user attempts to interact with the app, the apps are very slow, or in many cases, unresponsive. The main reason for this sluggish behavior of the apps is the hang bug problem.
3.3 Hang Problem

Hang problem is the major reason for the unresponsiveness of an app. It has also been a major threat to the software systems as it exists in several software systems such as web browsers, database servers and also in the android applications. Whenever these hang bugs emerge, the user does not get a response from the app within the expected time, and as a result, the entire system can freeze. Many times, the user might have to reboot the system to get it working again. Eventually the user will uninstall the app and might also write negative comments about the app in the Google Play store, which in turn will adversely affect the sales of the app. Since it is possible that the various apps developed might be very useful for the users for their day-to-day activities, these apps must be of high quality and should also load rapidly, as speed matters. [10].

3.3.1 Reasons for Hang

There are several reasons for the hang problem in mobile apps. A few of the well-known reasons are described below.

- **Environment**: Several hang bugs are caused due to the environment. For Example, the I/O disconnections like plugging out the network cable or shutdown of the network services can also cause hang bugs. [10]

- **Infinite Loop**: The hang bug is sometimes caused because the ‘while’ (true) loop termination condition cannot be satisfied, and as a result, exhausting all the CPU resources. [10].

- **Concurrency**: Most of the modern applications developed these days are multi-threaded or multi-processed so that these applications can run efficiently in this
multi-core architecture. As a result, the CPU consumptions are more, and concurrency becomes a major hang bug problem. The sub-categories are

- **Deadlock**: It is the most common hang bug problem in multi-processing systems. When the process or the thread waits for the system resource, which is being used by another process, the system is said to be in deadlock. This is called so because the process will not be able to change its state indefinitely, since the resources requested by the process are being used by another waiting process. [11].

- **Live lock**: These hang bugs are not similar to deadlock. This can be caused when an application waits for a non-existing thread or message. [10].

- **Design**: Poor design is the major reason for the hang bug problem. If heavy operations are performed in the main thread, the main thread will eventually take more than 100ms to execute the code as a result hang bug called soft hang is caused. The soft hang is discussed in detail below.

### 3.4 Soft Hang

The soft hang is the main focus of this research. The Android operating system will always have only one thread called the main thread or UI thread. The threads are the programmed instructions, which are being managed by the operating systems. The main thread is the most important thread in an application which is responsible for dispatching messages and for also interacting with the user widgets (Android UI toolkit). Since the main thread performs different user inputs such press a button, scroll etc., the main thread is always associated with a message queue. A looper and handler function will always
run on the main thread in order to insert or retrieve a particular message from the message queue. Only if the main thread is not busy, the handler recognizes which button is pressed by the user and is executed by the main thread. The main thread can execute only one event at a time. Therefore, whenever heavy operations like network access, database queries etc. are performed in the main thread, it blocks the entire main thread and the applications become unresponsive. These long running tasks can block the entire main thread and therefore no events are getting dispatched. So it is always a good idea to perform the long running operations in a separate thread called background thread. [12].

Therefore the algorithm proposed in this thesis helps to efficiently detect the long running APIs in the main thread so that these long running operations can be performed in a different thread in order to avoid the hang problem.
Chapter 4: Algorithm Design

4.1 Overview

In the algorithm design, there are mainly three tasks which take place in a synchronized manner. All the three tasks work in a synchronized manner in order to detect hang bugs. The first task is the Event Detector. The event detector acts as a timestamp. The start and finish of new events are known in the Event detector. The second task is the Trace Scheduler. The trace scheduler uses different methods to see how long it takes to complete the task. The time varies from one event to another. An event can finish in 10ms or 50ms or 100ms or 200ms etc. So, the trace schedule calculates the exact time spent by each event. The final task is the Trace Collector. The trace collector is responsible for collecting the stack traces of each event. If an event takes more than 100ms to complete the entire event as discussed before in detail in the introduction and background, there will be a hang bug. The trace scheduler notifies the trace collector if the event takes more than 100ms to complete the task. The trace collector then collects the stack traces of an event that are more than 100ms and the stack traces are analyzed to find the various causes of hang related problems. Thus this is the overall view of event detector, trace scheduler and trace collector.
4.2 Algorithm Development

4.2.1 Event Detector

The Event Detector acts as a timestamp. Whenever there is a new event, the event detector knows about the event. Once the event completes the entire task and stops, the event detector once again knows about the completion of the event. The event detector can be placed either in the main thread or it can be placed in the background thread. The main thread is always responsible for the execution of different events.

4.2.2 Trace Scheduler

Whenever an event takes more than 100ms to complete the task, the event detector synchronizes with the trace scheduler. The trace scheduler then calculates the amount of time required to complete the entire event. The amount of time to complete an event varies. There are different methods used in the algorithm in order to find an efficient way to calculate the amount of time spent by each event. The time of each event is calculated in order to find if there are any hang bug problems. If an event takes more than 100ms to complete the task then the application in the smartphone becomes sluggish or unresponsive. Therefore it is necessary to find the amount of time it takes to complete each event. There are two main methods used to calculate time of each event. The first method is called ‘sleep’. When the event starts, the thread sleeps, and it wakes up after 100ms to check whether the same event is being executing or if it is a different event. If it is the same event, then the trace scheduler notifies the trace collector in order to collect the stack traces of the event. If it is a different event then it checks the amount of time spent by the different event. By using the sleep method, we can save resources like CPU
utilization etc. The second method used is the timer task. Whenever there is a new event, the timer task schedules a timer with timeout. If the event takes more than 100ms to complete the task then the stack traces of the event are collected in order to find the long running events that cause hang bugs. If the event completes within 100ms the timer task ends or is cancelled. Thus these are the two methods used in the algorithm in order to find an efficient way to calculate the time taken by each event.

### 4.2.3 Trace Collector

Whenever an event takes more than 100ms to complete the event, the trace scheduler notifies the trace collector. The trace collector always responsible to collect the stack traces of events which lasts more than 100ms to complete. By seeing the stack traces, one can analyze the root cause of the hang bug. If an event takes more than 100ms to complete the task then the app becomes very slow and unresponsive. As a result, the app can hang or freeze. The trace collector helps to find the reasons for the app becoming slow. The heavy APIs might be running in the main thread and therefore the app might become slow. This can be found by analyzing the stack traces collected by the trace collector. The stack traces allows us to track the sequence of functions and also shows us the details of where it has stopped and also shows us the details of how it has reached to that point. In particular we use the function .get-stackTrace() in the background thread to collect the stack traces of APIs which runs more than 100ms in the main thread. The APIs which are occurred more frequently in the trace are considered to be the root cause of the hang.
4.3 Implementation

The implementation of the algorithm is mainly focused on heavy APIs that are executed on the main thread, thereby causing the hang. There are two ways in which the different algorithms are implemented. The first one is by implementing the event detector in the main thread. The advantages of implementing event detector in the main thread are

a. The start and finish of events are notified immediately only with a very limited delay in the event detector.

b. A more number of stack traces are collected for APIs which last more than 100 ms to complete the event. There are less events missed as the main thread is being given higher priority.

The main role of the main thread is to execute the user input actions such as press a button, scroll etc. and if the event detector is placed in the main thread, the code has to be light or else it will block the entire main thread as the main thread executes only one event at a time. By implementing the event detector in the background thread, it can be made sure that the main thread will not be blocked and can be light. The disadvantage is that the events will be notified to the event detector with more delay and as a result few events are missed and also only a few stack traces can be collected compared to the event detector placed in the main thread. All said and done, there are six algorithms developed in order to find an efficient way to detect the hang bugs. They are outlined below.

Case 1: Event Detector in the Main Thread (Synchronized Thread Sleep Method)

There are two threads used in the program, namely, main thread and a background thread. All the heavy operations take place in the background thread so that the main
thread is not blocked. As shown in the flow chart below, the event detector is in the main thread so if there is a new event then it is immediately known to the event detector. A ‘wait and notify and sleep’ method is used in the background thread. The background thread waits for a notification from the event detector. Whenever there is a new event, the event detector notifies the background thread. The background thread always sleeps for 100ms and wakes up after 100ms to see if the same event is being executed. If the same event is being executed, then it collects the stack traces in order to find the long running operations taking place in the main thread, which caused the hang bug problem. If the event finishes within 100ms, then the background thread starts to wait for the next event. By using the Thread Sleep method, the resource usage can be less and also since the event detector is used in the main thread, there will be less delay. As a result, the hangs are also not missed by the algorithm.
Figure 1: Event Detector in the Main Thread (Synchronized Thread Sleep Method)
Case 2: Event Detector in the Background Thread (Synchronized Thread Sleep Method)

In Case 2, the event detector is placed in the background thread. The advantage of placing the event detector in the background thread is that the main thread can be light, and will not be blocked. The disadvantage of placing the event detector in the background thread is that the event detector might encounter some delay in knowing the start and finish of the events. As a result, a few events can be missed, and also, only a few stack traces can be collected compared to the event detector placed in the main thread. Similar to Case 1, whenever there is a new event, the event detector is notified and the background thread sleeps for 100ms and will wake up after 100ms to see if the same event is being executed. If the same event is being executed after 100ms, the background thread executes the event and collects the stack traces of the event in order to see if there are any heavy operations running in the main thread. If the event finishes within the 100ms, the finish of the event is known in the event detector and the background thread will start to wait for the next event. By placing the event detector in the background thread, the overall main thread overhead i.e., the CPU and memory usage can be low.
Figure 2: Event Detector in the Background Thread (Synchronized Thread Sleep Method)
Case 3: Event Detector in the Main Thread (Wait and Notify Method and Timer Task in the Background Thread)

There are three main steps involved in this algorithm. The event detector is placed in the main thread, and it notifies the background thread whenever there is a new event. The background thread waits until a new event is being notified by the event detector. Once the event detector notifies the background thread regarding the new event, the background thread creates a runnable. The runnable is placed in the message queue of the timer thread. The timer thread executes the runnable placed in the message queue only if the event takes more than 100ms to complete. The stack traces of the events that last more than 100ms are collected and analyzed. It is analyzed in order to see if there are any heavy APIs running in the main thread which can cause the hang problem. If the event completes within 100ms the runnable is deleted from the message queue of the timer thread since there was no hang and thus no stack traces has to be collected. As a result the overhead can be high compared to Case 1 since in Case 1, thread sleep method is used, and also only one background thread is executed.
Figure 3: Event Detector in the Main Thread (Wait and Notify Method and Timer Task in the Background Thread)
Case 4: Event Detector in the Background Thread (Wait and Notify Method and Timer Task)

In this case, the event detector is placed in the background thread. As a result, the main thread can be light but there can be some delay since the start and finish of events can be known to the event detector with some delay. The background thread waits until a new event is being notified by the event detector. Once the event detector notifies the background thread regarding the new event, the background thread creates a runnable. The runnable is placed in the message queue of the timer thread and is executed only if the event takes more than 100ms to complete. Whenever an event lasts more than 100ms to complete, the runnable placed in the message queue of the timer thread executes the event. The stack traces of the events that last more than 100ms are collected and analyzed. It is analyzed in order to see if there are any heavy APIs running in the main thread, which can cause the hang problem. Whenever an event completes within 100ms the runnable placed in the message queue of the timer thread is deleted since there was no hang and thus no stack traces are required to be collected. The only difference between Case 3 and Case 4 is the position of the event detector. In Case 3, the event detector is in the main thread whereas in Case 4, the event detector is in the background thread. Since the event detector is in the background thread, there can be some delay, and only some APIs are collected, whereas the main thread overhead can be less compared to Case 3.
Figure 4: Event Detector in the Background Thread (Wait and Notify Method and Timer Task)
Case 5: Event detector in the main thread (Timer Thread in the background thread)

In Case 5, the event detector is placed in the main thread. Because of this, the start and finish of events are known immediately to the event detector, with less delay. In the previous Case 4, the wait and notify method was used in the algorithm but in Case 5, only the timer thread is used. As a result, whenever there is a new event, the event detector creates a runnable and is placed in the message queue of the timer thread. Whenever the event takes more than 100ms to complete the task, the runnable in the message queue of the timer thread executes the event. It also collects the stack traces of events that last more than 100ms. The stack traces are collected and analyzed in order to find if there are any heavy APIs running in the main thread that can cause hang. If the event is completed within the 100ms then the runnable placed in the message queue is deleted since the event does not cause any hang and no stack traces have to be collected. The delay can be less in Case 5, but the main thread overhead can be high since the event detector creates a runnable every time there is a new event. Since the event detector is in the main thread, it creates the runnable from the main thread. This can result in high overhead in the mainthread. The only difference between Case 4 and Case 5 is that in case 5 the event detector creates a runnable for every new event but in Case 4, the event detector notifies the background thread and the background thread creates a runnable.
Figure 5: Timer Task in Event Detector and Both in the Main Thread.
**Case 6: Timer Task in Event Detector and Both in the Background Thread.**

This case is similar to Case 5. The only difference is that the event detector is placed in the background thread. Since the event detector is placed in the background thread, there can be some delay. Whenever there is a new event, the event detector creates a runnable and is placed in the message queue of the timer thread and whenever the event takes more than 100ms to complete the task, the runnable placed in the message queue of the timer thread is executed. The stack traces of events that last more than 100ms are collected and analyzed in order to find if there are any heavy APIs running in the main thread that can cause hang. If the event finishes before 100ms then the runnable placed in the message queue of the timer thread is deleted since the event does not cause any delay and thus there is no need for collecting stack traces. Since the event detector is in the background thread, the main thread overhead can be less compared to the previous Case 5, but the delay is more than that compared to Case 5 since the start and finish of events can be known to the event detector with some delay.
Figure 6: Timer Task in Event Detector and Both in the Background Thread.
Chapter 5: Algorithm Evaluation

5.1 Evaluation

In this section a description of the experimental setup is provided, and also the algorithm is described as to how it detects the hang using various factors like delay, detection performance compared to baseline, main thread overhead, wholeapp overhead. Finally, comparisons of results from the six cases described in Chapter 4 are provided to show the efficiency of the algorithm chosen.

5.2 Experimental Setup

In order to evaluate the different cases of the algorithm, different android smartphone devices such as the LG NEXUS 4 with the android version 5.1.1 were used. The chosen algorithm was included in the folder containing the main thread source code and the algorithm was made to run in the android studio. Also an extra code was added on the onCreate of the main thread in order to start the algorithm when the app was first launched. When the source code of the app was run on the android studio, it automatically downloads the app on the mobile device as well. The android app on the mobile device was opened and various workloads were given to the app, in order to trigger the hangs. The response time of each event was taken into consideration in order
to find if there was any hang. If the event took more than 100ms to complete the event, then the app was considered to hang.

5.3 Hang Bug Detection

In this section, the results of the algorithm are shown when tested with the real world apps. Out of all real world apps tested, four apps: Stickercamera, Omninotes, BlockchainMerchant and UOIT/DC Booking apps are taken into consideration.

**Sticker Camera Android Application:** It is used to click a picture using the mobile device camera and then adjusting the face to the funny sticker. Different workloads are given to the sticker camera in order to trigger the app in order to produce the hang. The various workloads are as follow.

1. Open the Stickercamera app
2. Click the camera and take a picture
3. Save the picture.
4. Close the app.

**OmniNotes android application:** It is a note taking application. The various workloads given here are as follows:

1. Open the app
2. Click the photo option
3. Take a picture
4. Save the picture
5. Close the app
**BlockchainMerchant**: Accepts bitcoin as payment option. It is an ideal app for restaurants, cafes and all retail merchants who accepts Bitcoin as payment option. The various workloads are

1. Open the app
2. Enter any number as pin number
3. Enter the merchant name
4. Click the receiving payments option
5. Press the New payment address
6. Press scan button in the add payment address
7. Press back button twice
8. Close the app

**UOIT/DC Booking**: This app is used for booking a group study room at the University of Ontario Institute of Technology (UOIT) / Durham College (DC) in Oshawa, Ontario library. The various workloads are

1. Open the app
2. Click the policies button and press back button
3. Click the about button and press back
4. Press help button
5. Close the app

After performing various workloads to the apps, the stack traces are collected and analyzed in order to find the APIs that are causing the hang in the main thread. For all the six cases the stack traces are collected and analyzed in order to find which algorithm performs better and detects the hang bug efficiently. The various parameters which are
taken into consideration for testing the performance of the different cases are described as follows below.

5.3.1 Delay

The parameter delay is used to see how many events are missed. The delay is calculated as follows: “start dumping” means collecting stack traces of events which lasts more than 100ms and “dumping finished” means finish of the event. If there are stack traces collected after dumping finished, then there is some delay.

\[
\text{Delay} = \text{No. of stack traces collected after dumping finished} \times 20\text{ms}
\]

5.3.2 Number of Hangs Detected

The number of hangs detected is collected first without including the different cases, which is called ‘baseline’. Later on, after including each of the six cases, the number of hangs detected are collected in order to see if all the hangs were efficiently detected by these algorithms. The hangs detected by the six cases are compared with the baseline. If there are less hangs detected compared to the baseline then it shows that some events are missed and also there might be some noise if there are more hangs than the baseline.
5.3.3 Overhead Analysis

In this section the overhead is analyzed by taking the following factors into consideration: resource usage, CPU and memory traffic flow. First, the CPU and memory traffic flow for the entire app is analyzed. Then the overhead on the main thread is analyzed to see whether including this in each case causes any hangs bugs or high resource usage by the main thread.

5.3.4 Mainthread Overhead

The CPU and memory resource utilization of the main thread are calculated in order to see which algorithm performs better. The main thread overhead is calculated for cases in which the event detector is placed in the main thread. The main thread overhead is calculated to see if the event detector uses more resource and does the inclusion of these algorithms can block the main thread.

5.3.5 WholeApp Overhead

The CPU and memory utilization for the wholeapp are calculated in order to find which of the six cases utilizes less resource and are more efficient. The wholeapp overhead is also calculated to see if the algorithm doesn’t cause more resource usage.
Chapter 6: Results and Discussion

6.1 Delay

As shown in the Stickercamera app and the average of the entire four cases plot, the ‘Synchronized sleep with event detector in the main thread’ shows less delay compared to the Synchronized Sleep with the event detector in the background thread. This is because the event detector is in the main thread, and as a result, the start and finish of the events are notified in the event detector with less delay. Similarly, the ‘Synchronized Timer with event detector’ shows less delay compared to the ‘Synchronized Timer with event detector in the background thread’. The ‘Timer in the event detector, with the event detector in the main thread’ also shows less delay compared to the ‘Timer in the event detector, with the event detector in the background thread’. Overall, the ‘Synchronized Sleep with the event detector in the main thread’ has less delay of 20ms compared to the Synchronized Timer with the event detector in the background thread. Also, the ‘Synchronized Sleep with event detector in the main thread’ shows 25ms less delay than the ‘Timer in the event detector, with the event detector in the main thread’. The ‘Timer in the event detector, with the event detector in the main thread’ also shows 25ms less delay compared to the ‘Synchronized Timer with the event detector in the main thread’.
Figure 7: Delay Plot
6.2 Number of Hangs Detected

The following cases detect all the hangs compared to the baseline

1. Synchronized Sleep with the event detector in the main thread

2. Synchronized Sleep with the event detector in the background thread.

3. Timer in the event detector, with the event detector in the main thread.

All the three cases detect all the hangs compared to the baseline since the event detector is in the main thread as a result of which, the start and finish of events are known to the event detector with less delay.
Figure 8: Hangs Detected Chart
6.3 Main Thread Overhead

The main thread overhead is calculated in order to see if the inclusions of the different cases in the main thread can cause extra resource consumption. The main thread overhead of Case 1, i.e., ‘Synchronized Sleep’ and also the ‘Synchronized Timer’ are less compared to the ‘Timer in the event detector with the event detector in the main thread’ since for Timer in the event detector with the event detector in the main thread creates runnable from the main thread whenever there is a new thread. As a result the CPU and Memory consumption is higher. Compared to Synchronized sleep with event detector in the main thread and Synchronized timer with event detector in the main thread. Also the overhead of Synchronized sleep with event detector in the main thread is low compared to the Synchronized timer with event detector in the main thread. Overall Case 1 Synchronized sleep with event detector in the main thread performs better compared to Case 3 and Case 5 since the delay is less, and also the wholeapp overhead is also less compared to the other cases.
Figure 9: Main Thread Overhead Plot
6.4 WholeApp Overhead

The wholeapp overhead of ‘Synchronized Sleep with the event detector in the main thread’, is less than the ‘Synchronized Thread with the event detector in the background thread’. Similarly, there is only very slight difference in the overhead between the ‘Synchronized timer with the event detector in the main thread’ and the ‘Synchronized Timer with event detector in the background thread’. Finally, there is a small difference in the overhead between the ‘Timer in the event detector with the event detector in the main thread’, and the ‘Timer in the event detector with event detector in the background thread’. The Synchronized Timer has more overhead since two runnable are needed to execute the timer thread when events takes more than 100ms to complete the task.
Figure 10: WholeApp Overhead Plot
Shown below is a summary of the cases discussed, with key points stressed:

<table>
<thead>
<tr>
<th>CASES</th>
<th>DELAY</th>
<th>MAIN THREAD OVERHEAD</th>
<th>WHOLE APP OVERHEAD</th>
<th>HANGS DETECTED</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronized Sleep(M)</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>Synchronized Sleep(B)</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td>3</td>
</tr>
<tr>
<td>Synchronized Timer(M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Synchronized Timer(B)</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Timer in Looper (M)</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>3</td>
</tr>
<tr>
<td>Timer in Looper(B)</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Efficiency Chart Comparison

As seen, Case 1, ‘Synchronized Thread Sleep with event detector in the main thread’, performs better compared to other cases. All the parameters are low and efficient. Thus the algorithm performs efficiently in detecting hang bugs with less delay and overhead.

6.5 Conclusions

In this thesis, an efficient algorithm has been developed and tested with the real world apps in order to detect the root cause of the hang bugs. The algorithm differentiates itself from the previous work for the following reasons: the algorithm detects the hang bugs during the runtime when the user interacts with the app; further, it also detects the root cause of the hang bugs. The heavy APIs in the main thread are detected efficiently by the algorithm. In comparison, the algorithm shows high detection performance and low overhead. The algorithm also detects the root cause of the hang bug by the tracing the source code of the main thread. It eventually analyzes the execution path of those events which lasts more than 100ms in order to detect the heavy APIs in the main thread. In the future this algorithm can be extended to differentiate automatically between true hangs and false hangs as the false hang does not affect the application.
References


